Assignment 4:
Modify the *Linux Scheduler* to limit the CPU usage of a process family

HY345 – Operating Systems Course

Outline

- Background: The Linux Scheduler
- Modifying the Linux Scheduler
- Limiting the CPU usage of a process
  - Filter out the processes that their process family CPU time exceed the given limit at the last interval
    - Compute process’s execution time for the current time interval
    - Exclude processes from runqueue list so they cannot be chosen for execution
- Testing the new scheduler

Process Scheduling

- **Switching** from one process to another in a very short time frame
- Scheduler
  - When to **switch** processes
  - Which process to **choose** next
  - Major part of the operating system **kernel**

Linux Scheduler (in theory)

- Preemptive
  - Higher priority processes **evict** lower-priority running processes
- Quantum duration
  - Variable
  - Keep it as **long** as possible, while keeping good **response time**

Linux Scheduling Algorithm

- Dividing CPU time into *epochs*
  - In each epoch, every process has a specified **quantum**
    - Varies per process
    - Its duration is computed **when the epoch begins**
  - Quantum value is the **maximum** CPU time portion for this process in one epoch
    - When this quantum passes, the process is **replaced**
- Process **priorities**
  - Defines process's quantum

How it works

- At the beginning of each epoch
  - Each process is assigned a quantum
    - Based on its priority, previous epoch, etc
- During each epoch
  - Each epoch runs until its quantum ends, then replaced
    - If a process blocks (e.g., for I/O) before the end of its quantum, it can be scheduled for execution again in the same epoch

Linux Scheduler (in practice)

- Implemented in *linux-source-2.6.38.1/ kernel/sched.c*
- Main scheduler's routine is *schedule()*
- Data structures
  - policy (SCHED_FIFO, SCHED_RR, SCHED_RR)
  - priority (base time quantum of the process)
  - counter (number of CPU ticks left)

What happens in fork()

- The counter value is split in two halves
  - Half of the remaining clock ticks for the father
  - Half of the remaining clock ticks for the child

Runqeueue list

- A list with all runnable process
  - Process that are not blocked for I/O
  - Candidates to be selected by *schedule()* for execution

- *struct rq*
  - Defined in sched.h

The schedule() function

- Implements the Linux scheduler
- Find a process in the runqueue list for execution

- Invoked when a process is blocked
- Invoked when a process quantum ends
  - Done by *update_process_times()*
- Invoked when a process with higher priority than the current process wakes up
- Invoked when *sched_yield()* is called

schedule() in sched.c

```
/*
 * schedule() is the main scheduler function.
 */
asmlinkage void __sched schedule(void)
{
……….
……….
}
```

Actions performed by schedule()

- First it runs kernel functions that have been queued (drivers, etc)
  - run_task_queue(&tq_scheduler);
- Current process becomes prev
  - prev=current
- Next will point to the process that will be executed when *schedule()* returns

Round-robin policy

- If prev has exhausted its quantum, it is assigned a new quantum and moved to the bottom of the runqueue list

```
if (!prev->counter && prev->policy ==
   SCHED_RR) { prev->counter =
   prev->priority; move_last_runqueue(prev);
}
```

State of prev

- Wake up a process

```
if (prev->state == TASK_INTERRUPTIBLE
  && signal_pending(prev))
  prev->state = TASK_RUNNING;
```

  - Remove from runqueue is not TASK_RUNNING

```
if (prev->state != TASK_RUNNING)
  del_from_runqueue(prev);
```

Select next process for execution

- Scan the runqueue list starting from *init_task.next_run* and select as *next* the process with higher priority

```
p = init_task.next_run;
while (p != &init_task) {
  weight = goodness(prev, p);
  if (weight > c) {
  c = weight;
  next = p;
  }
  p = p->next_run;
}
```

Goodness

- Find the best candidate process
  - c=-1000 must never be selected
  - c=0 exhausted quantum
  - 0<c<1000 not exhausted quantum
  - c>=1000 real time process

```
if (p->policy != SCHED_OTHER)
   return 1000 + p->rt_priority;
if (p->counter == 0)
   return 0;
if (p->mm == prev->mm)
   return p->counter + p->priority + 1;
return p->counter + p->priority;
```

Empty runqueue or no context switch

- If the runqeue list is empty
  - No runnable process exists
  - Next points to the *init_task*
- If all processes in the runqueue list has lower priority than the current process *prev*
  - No context switch
  - *prev* will continue its execution

New epoch

- When c is 0 all processes in the runqueue list have exhausted their quantum
  - All of them have zero counter field
  - Then a new epoch begins

```
if (!c) {
  for_each_task(p)
  p->counter = (p->counter >> 1) + p->priority;
}
```

Context Switch

```
if (prev != next) {
   kstat.context_swtch++;
   switch_to(prev,next);
}
return;
```

Modifying the Linux Scheduler

- Schedule() function in sched.c
- Definitions in sched.h
- Add new fields in *task_struct* if needed

- *struct rq*
  - The main per-CPU runqueue data structure
  - Add fields in this struct for the scheduler

schedule() in 2.6.38.1

```
asmlinkage void __sched schedule(void)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;

…………
………….
```

schedule() in 2.6.38.1

```
  raw_spin_lock_irq(&rq->lock);
  pre_schedule(rq, prev);
  if (unlikely(!rq->nr_running))
  idle_balance(cpu, rq);
  put_prev_task(rq, prev);
  next = pick_next_task(rq);
  clear_tsk_need_resched(prev);
  rq->skip_clock_update = 0;

  if (likely(prev != next)) {
  sched_info_switch(prev, next);
  rq->nr_switches++;
  rq->curr = next;
  ++*switch_count;
  context_switch(rq, prev, next);
}
  raw_spin_unlock_irq(&rq->lock);
  post_schedule(rq);
```

Pick up the highest-prio task

```
static inline struct task_struct * pick_next_task(struct rq *rq) {
  const struct sched_class *class;
  struct task_struct *p;

  if (likely(rq->nr_running == rq->cfs.nr_running)) {
  p = fair_sched_class.pick_next_task(rq);
  if (likely(p)) return p;
  }

  for_each_class(class) {
  p = class->pick_next_task(rq);
  if (p) return p;
  }
}
```

In this assignement

- Limiting the execution time of a process
  - Or a process family as defined in assignment 3
- First, find if a process has a process family and a time limit
  - Scan all processes in the runqueue list
- If so, check if this process has a family that has exceeded the given time limit **in the current time interval**
  - So, also divide time in time intervals
- If so, remove this process from the runqueue list
  - So it will not be executed
  - Clone the runqueue list localy in this function for safety

Start from assignment 3

- Copy your code from assignement 3 and start with the new fields in tast struct and the two new system calls
  - You will use the setproclimit() for your tests
- Use qemu and same process to compile linux kernel and boot with the new kernel image

Pre-process and filtering in runqeue list

- Before *schedule()* selects the next process
- Clone the runqueue list rq for convenience to rq’
- Iterate the runqueue list rq. For each process p:
  - Check for root_pid!=-1 AND time_limit!=-1
  - If not, leave the process p into rq’
  - Else compute user+system time for process_family(p)
    - Only for the current time interval I
  - If user+system time for process_faimly(p) < time_limit(p), leave p into rq’
  - Else exclude p from rq’

Time Interval

- Divide time in time intervals
- Measure execution times for each time inteval

- Start a new time interval every *time_interval* milliseconds
- Add *prev_time* variable (you can add it wherever you prefer, e.g., in rq)
- If *current_time* > *prev_time* + *time_interval*
  - Start a new interval

At each new time interval

- Instead of zeroing user and system time per process
- Keep *prev_utime* and *prev_stime* per process
- CPU time of a process in the current interval is
  - *(utime-prev_utime)* + *(stime-prev_stime)*

- All the above are just **hints**
  - Feel free to implement this assignment in **any way** you prefer

Testing the modified scheduler (1/2)

- User-level programs that show the desirable behavior

- Test1
  - 1 billion multiplications, vary time limit
  - measure the effect of time limit in real time
  - Multiple programs with different time limit in parallel

Testing the modified scheduler (2/2)

- User-level programs that show the desirable behavior

- Test2
  - 1 billion multiplications per process, vary time limit and number of processes
  - measure the effect of time limit and number of processes in real time
  - Validate the process family limit

What to submit

1. bzImage
2. Only modified or created by you source files of the Linux kernel 2.6.38.1 (both C and header files)
3. Test programs and header files used in the guest OS for testing the modified scheduler
4. README with implementation details and experiences from testing

Gather these files in a single directory and send them using the submit program as usual

Good luck

Deadline: 20/1/2014